Metamorphic Testing of Deep Learning Compilers

DONGWEI XIAO, The Hong Kong University of Science and Technology, China
ZHIBO LIU, The Hong Kong University of Science and Technology, China
YUANYUAN YUAN, The Hong Kong University of Science and Technology, China
QI PANG, The Hong Kong University of Science and Technology, China
SHUAI WANG*, The Hong Kong University of Science and Technology, China

The prosperous trend of deploying deep neural network (DNN) models to diverse hardware platforms has boosted the development of deep learning (DL) compilers. DL compilers take the high-level DNN model specifications as input and generate optimized DNN executables for diverse hardware architectures like CPUs, GPUs, and various hardware accelerators. Compiling DNN models into high-efficiency executables is not easy: the compilation procedure often involves converting high-level model specifications into several different intermediate representations (IR), e.g., graph IR and operator IR, and performing rule-based or learning-based optimizations from both platform-independent and platform-dependent perspectives.

Despite the prosperous adoption of DL compilers in real-world scenarios, principled and systematic understanding toward the correctness of DL compilers does not yet exist. To fill this critical gap, this paper introduces MT-DLComp, a metamorphic testing framework specifically designed for DL compilers to effectively uncover erroneous compilations. Our approach leverages deliberately-designed metamorphic relations (MRs) to launch *semantics-preserving* mutations toward DNN models to generate their variants. This way, DL compilers can be automatically examined for compilation correctness utilizing DNN models and their variants without requiring manual intervention. We also develop a set of practical techniques to realize an effective workflow and localize identified error-revealing inputs.

Real-world DL compilers exhibit a high level of engineering quality. Nevertheless, we detected over 435 inputs that can result in erroneous compilations in four popular DL compilers, all of which are industry-strength products maintained by Amazon, Facebook, Microsoft, and Google. While the discovered error-triggering inputs do not cause the DL compilers to crash directly, they can lead to the generation of incorrect DNN executables. With substantial manual effort and help from the DL compiler developers, we uncovered four bugs in these DL compilers by debugging them using the error-triggering inputs. Our proposed testing frameworks and findings can be used to guide developers in their efforts to improve DL compilers.

CCS Concepts: • **Software and its engineering** → **Compilers; Software defect analysis**; • **Computing methodologies** → Neural networks.

Additional Key Words and Phrases: metamorphic testing; deep learning

*Corresponding Author: Shuai Wang (shuaiw@cse.ust.hk).
Authors' addresses: Dongwei Xiao, dxiaoad@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Zhibo Liu, zliudc@connect.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Yuanyuan Yuan, yyuanaq@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Qi Pang, qpangaa@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Shuai Wang*, shuaiw@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China.

2476-1249/2022/3-ART15 $15.00
https://doi.org/10.1145/3508035

ACM Reference Format:
Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang*. 2022. Metamorphic Testing of Deep Learning Compilers. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 15 (March 2022), 28 pages. https://doi.org/10.1145/3508035

1 INTRODUCTION

With the emerging demand to use machine learning (ML) and deep learning (DL) techniques in real-world scenarios, recent years have witted a tremendous deployment of ML and DL models in a wide spectrum of computing platforms, ranging from cloud servers to mobile phones and embedded devices. Deploying models toward such platforms is challenging, given the diversity of hardware characteristics (e.g., storage management, compute primitives) of various computing units, including TPUs, GPUs, CPUs, and even FPGAs.

To handle the complex deployment and explore the full potential of the computing platform for optimization, one highly promising trend is to employ *DL compilers* [7, 42, 64] which take a high-level DNN model specification (e.g., a model specification exported by TensorFlow [2]) and generates corresponding low-level optimized binary code for a diverse set of hardware back-ends. For instance, the de facto deep learning compiler, TVM [7], can offer performance boost competitive with manually optimized libraries and also smoothly compile high-level models into a diverse set of hardware back-ends.

Compiling high-level model structures into executable files denotes a complex pipeline that composes multiple layers of intermediate representation (IRs) and optimizations [7, 42, 64]. For instance, an input DNN model will be typically converted into graph-level IR where various frontend optimizations, including graph-level, node-level, and dataflow-level optimizations, are performed. Then, the optimized graph IR will be further converted into an operator-level IR, where compiler backend optimizations, including hardware-specific optimization and auto-tuning, are leveraged. These IR translations and optimizations require careful consideration, which imposes a high challenge to compiler engineers. In fact, recent works in the community have explored typical bugs that are constantly reported to the developers [68].

This work presents MT-DLComp [48], a systematic and automated testing framework for DL compilers. We aim to uncover logic errors in DL compilers that stealthily induce incorrect DNN executables. MT-DLComp performs metamorphic testing [9], a testing scheme that has achieved significant success in testing conventional software and also DNN models [65, 79, 87, 97]. In general, metamorphic testing is an invariant property-based testing method that relies on mutation rules referred to as metamorphic relations (MRs). Metamorphic testing alleviates the difficulty of determining the expected outputs of test inputs (which requires human annotation and is often prohibitively expensive) by verifying the target software's behavior consistency under MR-mutated test inputs. As a result, software testing becomes substantially more flexible.

We design two novel *semantics-preserving* MRs to transform DNN models and check DL compiler output consistency w.r.t. the original and mutated models. DNN models mutated by MRs exhibit largely distinct and much more complex network structures compared with the reference seed models, thus effectively stressing the compilation and optimization pipeline of DL compilers. Nevertheless, the mutated DNN models retain identical model predictions w.r.t. one or any model inputs. Hence, we envision that applying semantics-preserving MRs on DNN models and checking the consistency of compiled DNN executables can presumably evaluate the correctness of DL compilers. Furthermore, inspired by software delta debugging [95], we also propose an approach for minimizing error-triggering DNN models to facilitate debugging and error confirmation.

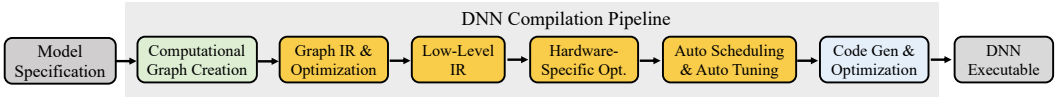


Fig. 1. The workflow of DL compilers.

Our large-scale evaluation encompasses four cutting-edge DL compilers, TVM [7], Glow [64], NNFusion [42], and Tensorflow XLA [76], which are all developed by industry giants. We use real-world DNN models as mutation seeds, including VGG11 [69], ResNet18 [25], and MobileNets [31]. MT-DLComp generates 1,500 mutated DNN models in total to test DL compilers. During approximately 20 days of testing, we detected 435 inputs that resulted in erroneous DNN executables generated by these DL compilers. We also found over 317 inputs that triggered compilation failures. With over 148 man-hours, we identified four bugs in those compilers that caused the faulty compilations. We have submitted all our findings to the developers of DL compilers; at the time of writing, several of these discoveries have been confirmed. In summary, we make the following contributions:

- This work introduces a new focus to launch systematic testing for DL compilers. We design testing methods to expose compilation bugs in DL compilers. The flagged bugs are highly critical and can largely change the behavior of compiled DNN executables.
- Our testing framework, named MT-DLComp, performs metamorphic testing to test DL compilers. Two novel MRs are designed to mutate DNN models and stress DL compilers. We also incorporate various design principles and optimizations to deliver effective testing and minimize error-triggering DNN models, which are usually enormous computation graphs with thousands of nodes.
- We tested four industry-leading DL compilers and find 435 error-triggering inputs. We have reported all the findings to the compiler engineers, and by the time of writing, typical defects have been promptly confirmed. With extensive manual effort, we identified four bugs (and one "false positive" due to numeric accuracy deviation) in DL compilers which resulted in *all* incorrect DNN executables found in our evaluation.

We release MT-DLComp and evaluation data of this research at [48].

2 PRELIMINARY

2.1 DNN Compilation

Fig. 1 depicts the general procedure of DNN model compilation. DNN compilation can be divided into frontend and backend phases [40], where each phase manipulates one or several distinct intermediate representations (IR).

Computation Graph. The inputs to DL compilers are high-level model descriptions, in particular, computation graphs that can be exported from DL frameworks like PyTorch [58]. A computation graph describes the network structure of a DNN model. DNN operators are represented as nodes in the graph, and connections between DNN operators are represented as graph edges. Computation graphs can be stored in various formats, but usually in ONNX format [1], so as to be acceptable inputs to DL compilers.

Frontend: Graph IRs and Optimizations. DL compilers usually convert DNN computation graphs into graph IRs at frontend. Network topology and layer dimensions encoded in graph IRs can facilitate optimizations such as operator fusion [7, 64]. For instance, after precomputing certain

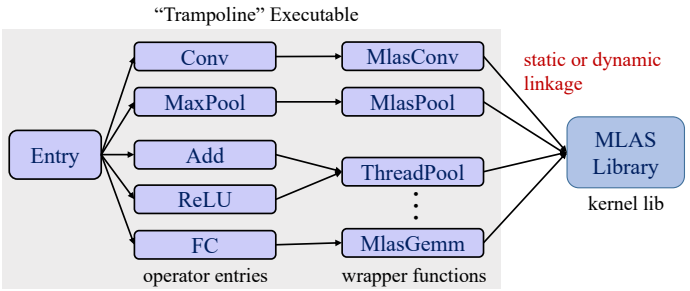


Fig. 2. The "trampoline" executable compiled by NNFusion or XLA.

components that can be statically determined, operator fusions and constant folding are conducted to identify mergeable nodes (e.g., small operators) from the graph IRs.

Backend: Low-Level IRs and Optimizations. Graph IRs only specify high-level inputs and outputs of each operator but do not restrict how each operator should be implemented. Low-level IRs are generated from graph IRs for hardware-specific optimizations. Low-level IRs usually involve memory-related operations. Hence, optimizations at this stage can include hardware intrinsic mapping, memory allocation, latency hiding, loop-related optimizations (e.g., loop tiles and unrolling), and parallelization [5, 7, 62, 94].

Backend: Scheduling and Tuning. Policies mapping a DNN operator to low-level code are called *schedules*. A compiler backend often needs to search a vast combinatorial schedule space and discover optimal parameter settings, such as loop unrolling factors. Halide [62] introduces a scheduling language describing many schedule primitives that can be used for manual and auto optimizations. Recent works explore launching auto-scheduling and auto-tuning to enhance the optimization quality [3, 7, 8, 49, 84, 101, 102]. These automated techniques are shown to alleviate manual efforts to decide schedules and to determine optimal parameters.

Backend: Code Gen and Optimizations. Low-level IRs are further compiled, using either just-in-time (JIT) or ahead-of-time (AOT) paradigms, to generate code for different hardware targets like CPUs and GPUs. Low-level IRs can be further converted into mature tool-chains like LLVM IR [37] or CUDA IR [52] to explore hardware-specific optimizations.

"Standalone" vs. "Trampoline" Code Gen Paradigms. The code generation paradigm of de facto DL compilers can be classified into two categories. Popular DL compilers like TVM and Glow compile optimized IR code into *standalone* executables. The compiled executables have no dependency on the underlying platform except some standard libraries, e.g., `libc`.

Alternatively, kernel libraries can be leveraged by some DL compilers, such as NNFusion [42] and XLA [76], to be statically linked with DNN executables. These kernel libraries are usually manually written by experts to fully exploit optimization opportunities offered by the underlying hardware. As illustrated in Fig. 2, each compiled executable contains a number of "trampolines" toward the kernel libraries, which provide the real implementation of DNN operators. It also contains some utility functions like `ThreadPool`. Our study shows that on Intel x86 platforms, executables are frequently linked with linear algebra libraries like Intel MKL and MlasGemm [46, 85]. The downside of this "trampoline" paradigm might be its heavy reliance on the underlying hardware platforms.

This work launches testing toward four production DL compilers, TVM, Glow, NNFusion, and XLA. The former two compilers generate "standalone" executables, whereas the latter two generate "trampoline" executables linked with kernel libraries. This illustrates the comprehensiveness of our testing. It is worth noting that during evaluation (Sec. 6), we find that compilers generating

"trampoline" executables contain negligible bugs; this could be due to the fact that compiling "trampolines" executables are easier than emitting a fully-optimized executable in the standalone format; see Sec. 6 for detailed evaluations and analyses.

2.2 Metamorphic Testing (MT)

Determining the correctness of DNN executables emitted by DL compilers is obscure, which may require human-annotated ground truth. In contrast, metamorphic testing benchmarks testing targets via **metamorphic relations** (**MRs**) without the need for ground-truth [9]. Each MR denotes a general and usually *invariant property* of the testing targets.

As an example, to test the implementation correctness of $sin(x)$, instead of knowing the expected output of arbitrary floating-point input x (which requires considerable manual efforts), metamorphic testing asserts whether the MR $sin(x) = sin(\pi - x)$ always holds when arbitrarily mutating x. A bug in $sin(x)$ is detected when input x and its variant $(\pi - x)$ produce inconsistent outputs. To date, metamorphic testing has achieved major success in detecting bugs in traditional software [38, 39, 41, 65, 73] as well as AI models [44, 71, 79, 87]. Our research further leverages metamorphic testing to benchmark DL compilers without requiring manual effort. To this end, we define two novel MRs to conduct *semantics-preserving* mutations toward the computation graph of DNN models.

3 TESTING ORACLE FORMULATION

3.1 Existing Efforts

Existing research has established a solid foundation for discovering and categorizing DL compiler bugs empirically [68]; defect-fixing pull requests (PRs) on GitHub are collected to characterize DL compiler bugs. In Sec. 8, we also present literature review about a large number of contemporary research efforts in testing DNN models and DL frameworks like TensorFlow and PyTorch [2, 58]. Nevertheless, a comprehensive and automated testing pipeline for DL compilers remains a critical missing piece in our understanding of the quality of popular DL compilers. There is a demanding need to gain insights into how much of a problem DL compilation is, given its indispensable role in providing systematic optimization to boost the adoption of DNN models on CPUs, embedded devices, and other heterogeneous hardware backends [4, 33, 47, 53, 54, 61, 89, 90].

3.2 Forming Testing Oracles Based on Invariant Properties

Testing DNN models and DL infrastructures often face the challenge of providing explicit testing oracle [59], because most computations launched by DNN models aim to provide an answer to a question for which no previous answer exists [96]. To date, invariant property-based testing methods, e.g., metamorphic testing and differential testing [59], are extensively leveraged to tackle the testing oracle problem [16, 79, 87, 97]. In short, invariant property-based testing schemes aim to test a software's *consistency*. Even if the correctness of actual outputs is unknown, it is still feasible to construct and check output consistency, the violation of which reveals a bug.

There are multiple potentially invariant properties that can be leveraged to construct testing oracles for DL compiler testing. In the rest of this section, we discuss three candidates to form testing oracles, where $\textbf{Oracle}_1$ and $\textbf{Oracle}_2$ are derived from differential testing and $\textbf{Oracle}_3$, the testing oracle used by MT-DLComp, is derived from metamorphic testing.

3.2.1 ***Oracle***$_1$*: Differentiate Outputs of Multiple DL Compilers.* Intuitively, we can compile a DNN model into DNN executables using multiple DL compilers available on the market, and further assert the prediction consistency of DNN executables with model inputs. Inconsistent model predictions indicate compilation defects. However, this oracle (i.e., $\textbf{Oracle}_1$) might not be desirable. DL compilers may implement different machine code generation strategies with different floating

number precision. As a result, DNN executables generated by different DL compilers can potentially produce inconsistent running outputs due to numerical accuracy instead of bugs.

Empirical Exploration. At this step, we study the accuracy deviation in terms of two production DL compilers, TVM and Glow, over image classification models provided by ONNX Zoo [56], including large-scale models like AlexNet [36] and Inception [75]. Note that these models are all commonly used in daily deep learning tasks. We compile the ONNX file of each model into a pair of DNN executables, one by TVM and the other by Glow. We then feed random images to each pair of executables and compare their output deviations (in terms of *confidence scores*). Out of 18 DNN models, we report that 16 models are compiled into executable pairs with non-zero output deviations. In particular, we find about one hundred outputs manifesting deviation greater than 10^{-2} (though the prediction labels are the same). These findings indicate a notable difference in the execution output from executables generated by different DL compilers. In addition, we also find that these DL compilers extensively employ Intel Streaming SIMD Extensions (SSE) instructions to compile DNN executables on x86 platforms, whereas instructions with low-precision floating numbers are picked when emitting executables on embedded devices [21]. In short, our preliminary study shows that violation of "consistency" across different DL compilers might not be due to bugs, whose root causes require extensive efforts for manual confirmation.

3.2.2 ***Oracle***$_2$*: Differentiate Outputs of DL Compilers and DL Frameworks.* Given that DL compilers translate DL models in high-level descriptions into DNN executables, another proposal of testing oracle (**Oracle**$_2$) is to assert behavior consistency of DNN models running on DL frameworks (e.g., TensorFlow) with DNN executables running directly on CPUs/GPUs. While ideally they should have identical behavior, we clarify that this oracle still suffers from floating-number precision loss and is also not desirable. Overall, DNN executables primarily leverage floating-point related machine code for computation. For instance, on x86 platforms, the Intel Streaming SIMD Extensions (SSE) instructions [18] are extensively used in DNN executables for floating-point related operations [7, 64]. In contrast, in DL libraries, floating-point numbers are often wrapped in objects with no or negligible precision loss.

Empirical Exploration. At this step, we compare DNN executables compiled by Glow, a production DL compiler, with corresponding DNN models running on TensorFlow. We compile all image classification models from ONNX Zoo using Glow. Then, we compare the execution outputs of DNN executables with their reference model running on TensorFlow using random images as model inputs. We report to see 49 cases with confidence score deviations that are larger than 10^{-2}, and thousands of cases with deviations over 10^{-3}. We even find a case with confidence score deviation 0.07. It is generally hard to analyze thousands of "inconsistencies" and distinguish true bugs from numerical accuracy deviations.

Besides TensorFlow, such deviations have also been observed when comparing DNN executables with DNN models running in other DL frameworks, including the ONNX runtime (`onnxruntime` [45]) and PyTorch. We have actually reported these findings to the DL compiler developers during our preliminary study. As pointed out by one key developer of Glow, it is not unexpected that DL compilers may manifest driftings in compiling floating-point related operations, which will be reflected in the outputs of compiled DNN executables.

3.2.3 ***Oracle***$_3$*: Inconsistency Between Multiple Semantics-Equivalent DNN Models.* This research formulates its testing oracle on the basis of metamorphic testing introduced in Sec. 2.2. To this end, we aim to design semantics-preserving MRs toward DNN models to assert the compilation consistency of a DL compiler over multiple semantics-equivalent DNN models. Let a DNN model m be mutated into another model m' which preserves the same semantics, DNN executables compiled

from m and m' by a DL compiler D should always make consistent predictions. This way, we stress a particular compiler D using different inputs. Overall, we implement the testing oracle (referred to as **Oracle**$_3$) of MT-DLComp following this scheme. In particular, Let m be a DNN model and a semantics-preserving MR be T, we mutate m into $m' = T(m)$ and check the following **Oracle**$_3$:

$$E[[D(m)]].Label = E[[D(m')]].Label \tag{1}$$

$$|E[[D(m)]].Score - E[[D(m')]].Score| < \delta \tag{2}$$

where $E[[\cdot]]$ executes a DNN executable (i.e., $D(m)$ and $D(m')$) and yields the output in terms of prediction label ($E[[\cdot]].Label$) and the associated confidence scores ($E[[\cdot]].Score$; a floating point number). We compare the predicted label and the confidence score yielded by $E[[D(m)]]$ and $E[[D(m')]]$ over model inputs (e.g., images).

The above testing oracle asserts if the prediction labels are identical in $E[[D(m)]]$ and $E[[D(m')]]$. On the other hand, instead of strictly asserting the confidence score equivalence in $E[[D(m)]]$ and $E[[D(m')]]$, we allow a small δ to tolerate numeric accuracy driftings due to compiler optimizations; see **Precision Losses Due to Optimizations** for clarifications. Our preliminary study shows that a small δ is sufficient (as most driftings are within 10^{-6}). δ is 10^{-4} in our current implementation, and users can configure MT-DLComp with different δ w.r.t. their particular usage scenarios.

Advantage of Oracle$_3$. Oracle$_3$, formed on the basis of metamorphic testing, is more desirable compared with **Oracle**$_1$ and **Oracle**$_2$. As clarified above, different DL compilers may use different compiling pipeline, and has compiler-specific optimizations that are not present in the other compilers. Similarly, DL frameworks use different methods for representing floating-point numbers with DL compilers. Hence, the output of DNN models from different compilers or DL frameworks may be different in terms of numerical accuracy. Therefore, even if the output is found to be different when cross-checking different DL compilers or frameworks, it is difficult to tell whether such difference comes from numerical accuracy or bugs. In fact, as we previously mentioned, most DNN models from ONNX Zoo, after compiled by different DL compilers or DL frameworks, exhibit distinct outputs, some of which even have a deviation of 10^{-2}.

Overall, **Oracle**$_3$ overcomes the inherent hurdles of **Oracle**$_1$ and **Oracle**$_2$, since our metamorphic testing approach compares each tested DL compiler with *itself*. For different DNN models being compiled, the tested DL compiler always uses the same set of platform-specific instructions, going through the same compilation pipeline.

Precision Losses Due to Optimizations. We clarify that DNN models may be optimized by the DL compiler to induce subtle precision losses. For instance, as a Glow developer mentioned (see Sec. 6.3), a DL compiler optimizer may remove some layers in the computation graph of a DNN model, inducing a slightly different result compared to that of the un-optimized version. Additionally, our observation shows that DL compilers may adaptively optimize a model. For instance, different optimization schemes may be applied, depending on the computation graph topology of a specific DNN model. This explains that semantics-equivalent DNN models may yield slightly different numeric outputs. However, we clarify that such driftings are small, usually around 10^{-6} according to our observation. A small δ (i.e., 10^{-4}) in Eq. 1 should compensate such driftings.

Note that this numeric drifting is due to DL compiler optimizations. That is, it is *orthogonal* to numeric deviations exposed in our preliminary studies on **Oracle**$_1$ (Sec. 3.2.1) and **Oracle**$_2$ (Sec. 3.2.2); recall those deviations are due to different implementations of DL compilers and DL frameworks. Using **Oracle**$_1$ or **Oracle**$_2$ cannot avoid the numeric driftings introduced by compiler optimizations as well. Even worse, **Oracle**$_1$ and **Oracle**$_2$ additionally suffer from numeric deviations

stemmed from different DL compilers and DL frameworks. As expected, the incurred precision losses when using **Oracle**$_1$ and **Oracle**$_2$ are generally higher than that in **Oracle**$_3$.

Table 1. Exploring deviations of three oracles using 300 mutated models. We exclude two mutated models triggering compilation failures.

	#deviations greater than **Oracle**$_3$	#deviations equal to **Oracle**$_3$	#deviations smaller than **Oracle**$_3$
Oracle$_1$	284	11	3
Oracle$_2$	283	3	12

At this step, we empirically assess precision losses when using these three oracles, whose results are presented in Table 1. In short, we mutate our seed DNN models (see details in Table 3), each for 100 times, and feed them (in total 300 models) to TVM, Glow, and TensorFlow. We setup **Oracle**$_1$ by differentiating outputs of TVM and Glow. **Oracle**$_2$ is used when differentiating outputs of TensorFlow and Glow. We test Glow using **Oracle**$_3$. As in Table 1, out of in total 300 inputs, there are 284 cases where the deviations, exposed when using **Oracle**$_1$, are greater than that of **Oracle**$_3$. Similarly, the deviations exposed by **Oracle**$_2$ are larger than that of **Oracle**$_3$ for 283 cases. Even worse, we found 166 deviations exposed when using **Oracle**$_1$ are over four times larger than that of **Oracle**$_3$. In **Oracle**$_2$, we find 208 cases manifesting deviations at least four times larger than that of **Oracle**$_3$. In summary, it is seen that **Oracle**$_1$ and **Oracle**$_2$ largely suffer from numeric accuracy deviations, and therefore, **Oracle**$_3$ should be desirable in testing DL compilers.

3.3 Alternative Approach: Fuzz Testing

Fuzz testing, a security testing technique, is generally used to detect software's obviously abnormal behavior (e.g., crashes or hangs) that are often related to memory access violations. Fuzz testing usually employ extreme test inputs, and bugs exposed by fuzz testing are noticeable to users since when the software crashes or hangs, users can immediately know that the software goes wrong. On the contrary, all three testing oracles mentioned above aim at exposing stealthy **logic errors**. With logic errors, DNN models can be successfully compiled by DL compilers, but stealthily, give incorrect output. The logic errors are much harder to be noticed by users since the compilation and execution do not exhibit obviously abnormal behaviors. Overall, fuzz testing, with no requirement on explicit oracles, is generally easier to set up during in-house development. Hence, fuzz testing is not considered in this research. In contrast, we focus on detecting logic bugs, which are much more stealthy and require a sophisticated-designed observer (i.e., testing oracles) for detection.

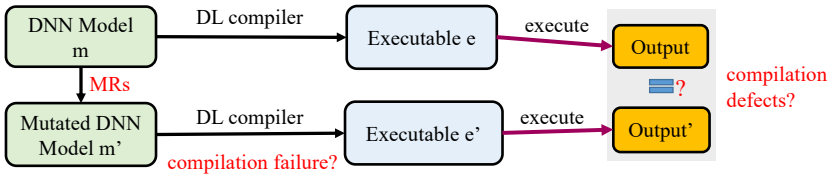


Fig. 3. Workflow of MT-DLComp. While we mainly focus on detecting logic bugs in DL compilers (i.e., "compilation defects"), we also record possible "compilation failures" occurred during testing.

4 METHODOLOGY

Fig. 3 depicts the workflow of MT-DLComp. Given a seed DNN model m, we first perform two MRs to mutate m iteratively; we thus generate a set of mutated $m' \in M$ which exhibit identical

functionality with m. In particular, Sec. 4.2 presents the design of a novel universal opaque condition (UOC) to generate m' that retains identical functionality with m under any inputs. We also design a fine-grained *per-input* opaque condition (PIOC), whose mutated m' shall yield identical prediction with m regarding a specific input i. These two schemes are employed jointly to iteratively mutate m and maximize chances of triggering inconsistency in DL compilers. Then, the seed model m will be compiled into an executable e using the target DL compiler C, and each mutated model $m \in M$ is also compiled into executable e' using C. Given that the adopted DNN models are for daily image classification tasks, we check the prediction consistency of e and e', in terms of prediction labels and confidence scores, over test images using our oracle (**Oracle**$_3$ in Sec. 3). We collect m', as an error-triggering DNN model, in case inconsistency is found by comparing the prediction outputs of e and e'.

We note that the mutated DNN models are generally enormous (see statistics in Table 3). Thus, it is challenging for developers to debug their DL compiler with an error-triggering DNN model m'. Accordingly, we extend the standard delta debugging [95] technique into a DNN computation graph-aware form to reduce the computation graph of a DNN model so as to get a minimal model that still triggers the error (see Sec. 4.3). Finally, we conduct manual investigation to explore root causes of the errors and aggregate the information to deduce empirical findings on DL compilers; see details in Sec. 4.4.

Study Scope. As depicted in Fig. 3, we capture the following two kinds of issues in this study:

- **Compilation defects** represent deviant results when we execute and compare the prediction outputs of DNN executables e and e' using **Oracle**$_3$ defined in Sec. 3. Deviant outputs imply the semantics of the input DNN model m' is ill-compiled into e', denoting a compiler bug.
- **Compilation failures** represent obvious errors, e.g., crashes or exceptions thrown by the DL compilers. According to our observation, this might be due to conflicts in certain DNN optimization combinations.

This work aims to uncover *compilation defects* that result in incorrect DNN model compilation. Overall, compilation defects can emit erroneous DNN executables silently. Following conventions in software testing literatures, the rest of this paper will frequently refer to compilation defects as *"stealthy logic bugs"* in DL compilers. Note that our main focus is not *compilation failures*, given that fuzzing or regression testing may likely find such failures during in-house development. Nevertheless, we still record and report all compilation failures we encountered during evaluation.

4.1 Seed Selection

Despite the growing adoption of DL compilers in production [4, 33, 47, 53, 54, 61, 89, 90], our preliminary study shows that popular DL compilers may still struggle in compiling certain natural language processing (NLP) models. NLP models may take some recurrent structures, such as RNN and long short-term memory (LSTM) [30], to process sequential inputs. DL compilers like TVM show immature support for these operators. Similarly, some newly developed models may contain operators that are currently only supported by certain DL frameworks. For instance, Nair et al. [50] compute Fast Fourier Transformation (FFT) by using a TensorFlow-based FFT layer, which, according to our observation, is currently not supported by DL compilers.

We agree that *rarely-seen DNN models* are in general useful to stress DL compilers. However, we find that de facto DL compilers are primarily designed to compile DNN models used in daily image analysis tasks [40]. Accordingly, our study mainly focuses on uncovering DL compiler defects when compiling *daily DNN models*. To this end, we use three image classification models from ONNX Zoo [1] as the seeds for metamorphic mutations (see details in Table 3). We thus assume that compiling our seed models should be reliable, as all seeds are common image classification

models. In other words, violation of our testing oracle implies compilation bugs, rather than some unsupported features. We present further discussions regarding seed selection in Sec. 7.

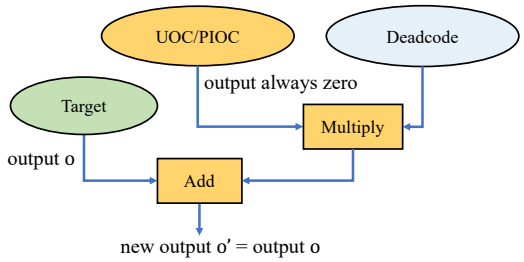


Fig. 4. Mutating DNN computation graph with UOC and PIOC.

4.2 Mutating DNN Computation Graph with Two MRs

Fig. 4 illustrates the high-level overview of our mutation over the DNN computation graph. Note that each node of this graph denotes one or a set of DNN operators. The Target node denotes a DNN operator in the input DNN model whose output is o. We propose two MRs, which insert universal obscure conditions (UOC) or per-input obscure conditions (PIOC) into the computation graph. From a holistic perspective, the inserted UOC/PIOC nodes denote an "if statement" with an *obscure* predicate that is generally difficult for DL compilers to optimize out during static compilation, but will always be executed as zero during runtime. In fact, our manual investigation shows that none of our inserted UOC/PIOC nodes are directly optimized out by DL compilers. See details of UOC and PIOC in Sec. 4.2.1 and Sec. 4.2.2, respectively.

In addition to inserting the UOC/PIOC nodes that complicate the DNN model, another key benefit is that we are able to freely include arbitrary DNN operators on the basis of UOC/PIOC without influencing the final output of DNN computations. As illustrated in Fig. 4, UOC/PIOC enables the insertion of arbitrary "Deadcode" nodes, given that the output of Deadcode will be multiplied with the output of the UOC/PIOC nodes, which nullifies the computation taken place in Deadcode. In the implementation, we randomly select a set of DNN operators to form Deadcode; see details in Sec. 4.2.3. Finally, the output o of our Target node will be added together with the output of the Multiply node, retaining the original output o.

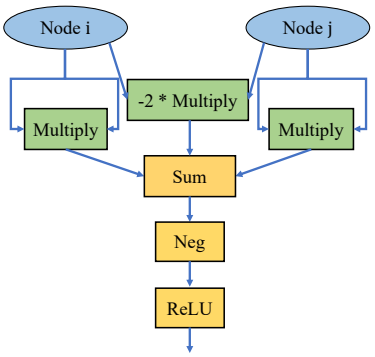


Fig. 5. Simplified view of UOC.

4.2.1 MR_1: Universal Obscure Condition (UOC). We first design an MR, by introducing a sequence of DNN operators that will *always* yield zero as the final output during the runtime. This scheme is referred to as a *universal* obscure condition (UOC), as it always yields zero despite different values of model inputs. The sequence of DNN operators introduced by UOC, however, is complex and hard to be optimized out, thus stressing the compilation and optimization phases of DL compilers.

Fig. 5 depicts our implementation of UOC. Given two nodes i and j on the computation graph of a DNN model, we employ a classic number-theoretic construction, i.e., $i^2 - 2 \times i \times j + j^2 \geq 0$, such that i and j are the outputs of node i and node j, respectively. The output of the Sum node will be first fed to a Neg node, then linked to a ReLU node. Note that ReLU denotes

a non-linear activation function defined as $f(x) = max(0, x)$. As a result, the final output of this DNN operator sequence is:

$$\mathsf{ReLU}(-1 \times (i - j)^2)$$

It is easy to see that no matter what value the node i and j output, the UOC will always output *zero*. More importantly, we find that DL compilers are generally unable to reason number-theoretic construction like $i^2 - 2 \times i \times j + j^2$ and decide its output always ≥ 0. Thus, the obscure sequence of DNN operators introduced by UOC will not be treated as "deadcode" and easily eliminated.

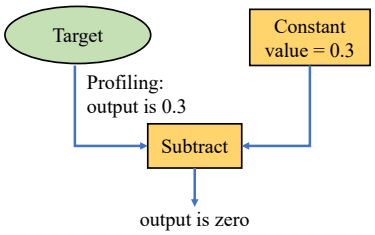


Fig. 6. View of PIOC.

4.2.2 MR_2: Per-Input Obscure Condition (PIOC). In addition to UOC, we further illustrate PIOC in Fig. 6. Overall, this scheme involves a profiling phase to acquire runtime information w.r.t. an input i. Then, PIOC introduces a condition node by mutating the target model, as illustrated in Fig. 6, whose computation output would be zero in case the specific input i is fed to the model. In general, PIOC has the following two steps:

Profiling Phase. We first deploy the to-be-mutated model m on `onnxruntime` [45], and conduct profiling to record its intermediate data. In particular, given a node n on the computation graph of m, we record the output o of node n when the model's input is i. Node n can be randomly selected from the computation graph of m.

Mutation Phase. For the specific input i and node n on the DNN model, the goal of PIOC's mutation phase is to use node n and its recorded output o to construct a sequence of DNN operators whose final output is zero w.r.t. input i. To achieve that, we introduce a DNN node that yields a constant output o, as shown in "Constant" in Fig. 6. We then place a subtraction node whose inputs are from the outputs of node n and the created constant node. It is easy to see that the output of the subtraction node will be zero w.r.t. input i. This way, we mutate the computation graph of the DNN model by adding extra nodes and edges without affecting the prediction output.

Alternative Realizations of UOC and PIOC. We admit that there exist many other possible realizations of the UOC and PIOC schemes. For instance, to implement UOC, we can utilize the following formula,

$$\mathsf{ReLU}(\sin x^3 - \ln(\frac{\sin^2 x}{2 + \cos x} + \cos^3 x + 10))$$

which, though highly obscure, will always output zero. Note that the maximum value of $\sin x^3$ is 1, and $\sin^2 x/(2 + \cos x)$ cannot be negative since its numerator is non-negative and its denominator is always positive. $\cos^3 x$ is always greater or equal to -1. Therefore, the value of $ln(\cdot)$ is no less than $\ln(0 - 1 + 10) = 2.30$. Thus, the input of ReLU is negative, meaning that the overall output is zero. Similarly, we can extend the current realization of PIOC, by inserting another Negation operator over the output of the inserted subtraction node; it is easy to see that the final output is still zero w.r.t. input i.

Despite the fact that there are numerous other possible implementations of UOC/PIOC, we deem our current realization practical and efficient for two reasons. First, the current implementation of UOC/PIOC only involves basic operators, such as Sum and Multiply, which are smoothly supported by DL compilers. Other realizations that contain more sophisticated operations may not be supported by DL compilers. Second, these basic DNN operators incur relatively low cost, whereas some

operators like *sin*, *log* are more computationally expensive. We leave it as a future work to explore other practical realizations of UOC/PIOC and compare the effectiveness of different realizations.

We also clarify that it is *not* necessary to design too complex realizations. Soon in Sec. 4.2.4, we will clarify that MT-DLComp iteratively mutates a seed model, meaning that with relatively lightweight UOC/PIOC realizations applied for hundreds of iterations, the mutated models can become highly complex; see statistics of our mutated models in Table 3.

4.2.3 Garbage Node Insertion. Both UOC and PIOC schemes introduce a number of extra DNN operators and edges on the computation graph without affecting the prediction outputs. More importantly, by inserting the aforementioned DNN operators into the computation graph, we actually reveal opportunities to insert extra "garbage nodes" to further complicate the computation graph of the DNN model. As illustrated in Fig. 4, "Deadcode" would not affect the computation, given that their output will be multiplied with the output of UOC/PIOC, which would evaluate to zero. The newly-inserted DNN operators and edges will be compiled and optimized together with the rest components of the computation graph, fruitfully complicating the computation graph and stressing the DL compilers.

In general, we use five different DNN operators to insert garbage nodes, including Conv, Dense, and three arithmetic operators (Add, Subtraction, Multiply). Each time we randomly select one DNN operator to serve as the "deadcode" and insert it together with the UOC/PIOC components into the computation graph.

4.2.4 Iterative Mutation. Starting from a seed DNN model M, we iteratively mutate M into a sequence of mutated models $M_1, M_2, \cdots, M_n$, where M_k is generated by *randomly* selecting either UOC or PIOC to launch mutation over M_{k-1}. This way, we gradually accumulate a sequence of mutated models sharing identical functionality. Each time when performing UOC/PIOC mutation, we need to first decide a target node (the "Target" in Fig. 4) for mutation. For the current implementation, we randomly select a position to mutate.

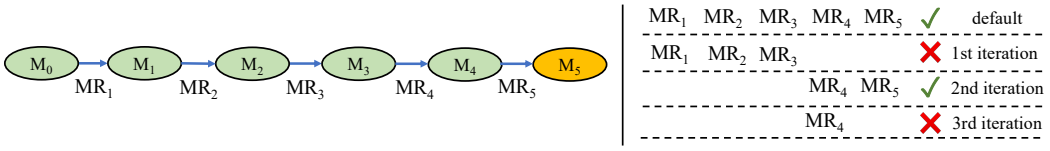
4.3 Reducing Error-Triggering Models

Given inputs that violate our testing oracle, we seek to further identify bug-inducing points, which can facilitate developers for debugging and patching. Our adopted DNN models, such as ResNet18 and VGG11 (see Table 3), are complex DNN models with a large number of DNN operators. Moreover, our mutations, by iteratively inserting a number of extra DNN nodes, typically make the mutated DNN models contain tens of thousands of operators and edges. Despite the mutation method's promising strength to stress DL compilers, we note that it becomes formidable for developers to debug DL compilers with the enormous mutated models.

Inspired by software delta debugging [95], we aim to reduce error-triggering DNN models. In short, we aim to minimize an error-triggering DNN model M by reducing nodes in its computation graph that are not needed to reproduce the bug, until a minimal graph is found. We then provide developers with M_{min}, a much smaller error-triggering DNN model, for manual debugging.

Fig. 7 depicts a *simplified* procedure of our approach at this step. Given a seed DNN model M_0, we iteratively apply mutations (e.g., using either UOC or PIOC) until detecting an error-triggering model M_5. We then launch the delta-debugging procedure, in which the 1st iteration, as shown in Fig. 8(b), divides the sequence of leveraged MRs into half. We apply the first half of the mutations (MR_1, MR_2, MR_3) over M_0, which does not trigger errors. Then, in the 2nd iteration, we apply the second half of mutations (MR_4, MR_5) to mutate M_0. This iterative exploration will proceed until we localize a minimal mutation set over M_0 that can still trigger the original error.

The above procedure is conceptually similar to binary search when the error-triggering mutations are contained in only one half. Nevertheless, we clarify that delta debugging provides a "divide and



(a) Iteratively mutating a model until triggering errors at M_5.

(b) Delta-debugging. ✓ denotes still trigger errors while ✗ denotes not.

Fig. 7. Minimizing error-triggering DNN model M_5. Each "MR_k" stands for a UOC or PIOC mutation. To ease presentation, we simplify the procedure: after three reduction iterations, we identify M_{min} with applying only MR_4 and MR_5.

conquer" procedure to tackle the case where the error is due to the interaction of certain mutations in *both halves*. That is, in case the bug is triggered by the combination of some mutations in the left half (MR_1, MR_2, MR_3) and the right half (MR_4, MR_5), MT-DLComp will launch further search within the right half while retaining the left one (MR_1, MR_2, MR_3), and similarly, search within the left half while retaining the right one (MR_4, MR_5). To ease the presentation, this "divide and conquer" procedure is omitted in Fig. 7. Interested readers may refer to the delta-debugging paper [95] and our released artifact [48] for details.

Note that when performing mutation MR_k over model M_{k-1}, DNN operators inserted by MR_k may depend on the outputs of DNN operators introduced by previous mutations. That is, certain mutations (MR_i and MR_j) have dependency and should be used together. Hence, during the delta-debugging iterations, when applying mutation MR_j, we include all its dependent MRs, and such a dependency relationship may unavoidably include many MRs. Nevertheless, given certain operators introduced by MR_k yield constant outputs (e.g., each UOC operator sequence always yields zero), we replace operators yielding constants with a single constant DNN operator to further reduce the complexity at this step.

4.4 Manual Inspection

Given minimized error-triggering DNN model M_{min}, we seek to further localize the buggy implementation in the DL compiler's source code. To achieve that, we first localize the erroneous DNN operator, then the error-inducing IR statements during compilation, and finally, the buggy modules in the DL compiler's source code. We now elaborate on each step.

Let e_{min} be the DNN executable compiled from M_{min}, we first use a popular decompiler, IDA-Pro [29], to disassemble e_{min} and collect all assembly function information in the assembly code. Then, by reading the assembly function name, we can map each assembly functions to the DNN operators. For instance, a Conv operator will be compiled into an assembly function with name `libjit_conv2d_f_[id]_specialized` by Glow, where `[id]` is a number assigned by Glow.[1] After that, we execute e_{min} and check whether the input values of each assembly function are correct. To do so, we hook the entry point of every assembly function with the GDB debugger, and compare the function inputs (memory regions on heap) with the inputs of its corresponding DNN operator collected by running M_{min} on the ONNX runtime `onnxruntime` [45].

When certain inconsistency over a DNN operator o is localized, we then export and inspect the IR programs emitted by the DL compiler when compiling M_{min}. We compare IR programs with

[1]Note that a DNN operator may frequently be fused with its neighbor operators due to optimization (Sec. 2.1), we are *not* expecting to have one-to-one mapping between the assembly function with the high-level computation graph of M_{min}. Nevertheless, we clarify that some key DNN operators, e.g., the Conv operator, are always mapped to individual assembly functions. We rely on these critical DNN operators to align assembly functions with the high-level computation graph.

M_{min} in the ONNX format, particularly toward operator o and its neighbor operators. We compare the *connectivity* of o (and its neighbor operators) in the IR program with their corresponding node and edges on the computation graph. We aim to decide if a certain operator in the IR is connected to incorrect nodes. This way, we successfully identified compiler bugs in IR translation; see details in Sec. 6.1. Note that a DNN operator may frequently be fused with its neighbor operators due to optimization (Sec. 2.1); as a result, it is vital to investigate as many neighbor operators as possible on the computation graph.

Given IR statements that result in erroneous compilations, e.g., an erroneous edge in the DNN computation graph, we iterate the source code of the tested DL compiler to localize modules that contribute to the compilation or optimization of those erroneous IR statements. We then pack the findings, including the error-triggering inputs, erroneous DNN operators, incorrect IR statements, and likely buggy modules in DL compilers, and forward them to the developers for confirmation.

Overall, this step is costly: examining and checking all suspicious findings took two analysts about 128 man-hours. Each analyst has an in-depth knowledge and experience in compilation techniques, DL compilers, and software testing. This way, we ensure the accuracy of our study and the credibility of our findings to a great extent. We have reported our findings to the developers, and some typical cases have been promptly confirmed and to be fixed (see Sec. 6). Moreover, with about 20 man-hours, we locate 4 buggy code fragments in DL compilers that lead to all violations of our testing oracles (see discussions in Sec. 6.2).

Table 2. DL compilers evaluated in our study.

Compiler	Publication	Developer	Version in git commit or docker version (commit date)	Code Gen Paradigm
TVM [7]	OSDI '18	Amazon	80de1239e (2021-09-25)	Standalone
Glow [64]	arXiv	Facebook Inc.	0abd13adc (2021-09-21)	Standalone
NNFusion [42]	OSDI '20	Microsoft	nnfusion/cuda:10.2-cudnn7-devel-ubuntu18.04 (2021-09)	Trampoline
Tensorflow XLA [76]	NA	Google	7b596c44 (2021-10-03)	Trampoline

5 STUDY SETUP

Our proposed testing framework, MT-DLComp [48], is implemented in Python, with 3,239 LOC (measured by `cloc` [12]). In addition to the testing framework, we also spend extra engineering effort into extending a tool onnx-tf [77], which converts DNN models from the ONNX format into the TensorFlow format. Note that XLA, the DL compiler released by Google, only accepts models in the TensorFlow format. However, we find that XLA still struggles to accept the outputs of onnx-tf. We spend extra engineering effort into tweaking the output of onnx-tf with about 100 LOC. We now discuss the details of our study setup.

DL Compilers. Table 2 reports the DL compilers tested in this research. We build the Github versions of these four compilers, whose corresponding Github commit numbers are also reported. These four DL compilers, to the best of our knowledge, represent the best DL compilers with broad application scope and support to various hardware platforms. All DL compilers are studied in the standard setting. In all, these DL compilers are designed for an "out-of-the-box" usage. Note that these DL compilers will apply **full optimization** by default.

Sec. 2.1 and Fig. 2 have clarified the high-level workflow of DL compilation and two code gen paradigms. TVM and Glow offer the standalone code generation paradigm, where DNN models are converted into high-level/low-level IRs to go through machine-dependent and machine-independent optimization phases. The outputs of these include the entire highly-optimized assembly instructions. TVM and Glow are marked as "standalone" in Table 2. In contrast, as noted in Fig. 2, NNFusion and

XLA are in the category of generating "trampoline" executables. We find that on x86 platforms, executables compiled in this paradigm are linked with optimized linear algebra libraries like Intel MKL and MlasGemm [46, 85].

NNFusion Defects. Among these four representative DL compilers, we report to encounter a considerable number of failures and exceptions (over 1,000) thrown by NNFusion when compiling the (mutated) DNN models. These thrown exceptions, to our knowledge, are mostly due to immature optimizations on even common DNN operators. Therefore, in the following sections, we skip testing NNFusion.

Statistics of DNN Models. In this research, we pick three representative DNN models as the seed models. We pick VGG11 and ResNet18, two large-scale computer vision models, as two of the seed models for mutation. Both models are widely used in daily image classification tasks. To possibly explore corner cases of DL compilation behaviors, we also select another model, MobileNets, with lower complexity. MobileNets is commonly used in mobile devices for image classification tasks.

Starting from one seed DNN model M (e.g., VGG11), we iteratively mutate M for N times: the mutated model M_k at the kth iteration serves as the input for the $k+1$th mutation. For the $k+1$th iteration, we randomly decide to use either UOC or PIOC to mutate M_k, and we randomly select a position on the computation graph of M_k for mutation. N is set as 1,000 in our experiments. Also, for each seed model M, we use five seed numbers and launch five series of iterative mutations. Given in total three seed DNN models (ResNet18, VGG11, MobileNets), we generate in total $3 \times 5 \times 1000$ mutated DNN models. To speed up the testing, we collect the first out of every ten mutated models, i.e., the 1st, 11th, 21st, … model, and feed them to the DL compiler for testing. This way, we use in total 1,500 mutated DNN models for testing.

Table 3. Statistics of DNN models. Each model will be mutated for 5×1000 times using five different random seeds. For "Average Mutated", we compute the average statistics of all mutated models. For "Largest Mutated", we measure the statistics of the largest mutated model.

Model	Seed			Average Mutated			Largest Mutated		
	#Parameters	#Nodes	#Edges	#Parameters	#Nodes	#Edges	#Parameters	#Nodes	#Edges
ResNet18 [25]	11,169,162	49	100	12,224,282	6,258	9,343	14,373,413	12,796	19,047
VGG11 [69]	28,144,010	28	51	39,693,434	6,376	9,430	62,989,804	13,130	19,372
MobileNets [31]	2,219,626	99	215	10,243,529	6,708	9,828	22,268,817	13,373	19,642

Table 3 reports statistics of these (mutated) DNN models. We report the complexity of DNN models in terms of #operators and #parameters in the computation graph. In addition to the seed model (the **Seed** column in Table 3), we also measure the complexity of the largest model (the **Largest Mutated** column): given that we iteratively mutate each seed model, the last mutated model denotes the largest one used for testing. The **Average Mutated** column reports the average statistics of all mutated models.

Preparing Validation Inputs. Note that to cooperate with PIOC, the validation input i needs to be the same as the one used in the profiling phase so as to force the output of each PIOC instance to be zero. We use images from CIFAR10 [35] as the validation inputs for running the compiled DNN executables.

6 EVALUATION

Table 4 provides an overview of our findings. It is worth noting that TVM by default enables full optimizations (noted as "TVM -O3" in Table 4). However, we encounter a large volume of compilation errors for this setting; the TVM developers suggest that the full optimization "messes up" the intermediate data structures used during compilation (see Sec. 6.3 for their reply). We thus

resort to using a lower level of optimization (denoted as "TVM -O2" in Table 4) and re-run the testing on TVM.

Table 4. Result overview. As clarified in Sec. 5, we exclude NNFusion and TVM (with full optimization enabled) since these two fail to compile too many models. We give case study about compilation failures in Sec. 6.1. Here "Logic Error" denotes stealthy compilation defects (incorrect DNN executables).

DL Compiler	#Errors Exposed		Logic Errors Exposed via Different Seeds		
	Compilation Failures	Logic Error	ResNet18	VGG11	MobileNets
TVM -O2	0	202	0	127	75
Glow	69	233	45	38	150
XLA	0	0	0	0	0
Total	69	435	45	165	225

Out of in total 1500 test cases, we find 69 compilation failures. While most models can be successfully compiled and executed, 435 (233+202) compiled executables manifest inconsistent prediction outputs with the executable compiled from the seed model, violating our testing oracle. XLA outperforms the other two DL compilers, with zero defects exposed. We also characterize errors exposed by different seed models in Table 4. Overall, it is seen that all three DNN models can trigger defects of TVM and Glow. ResNet18 and VGG11 have 10× more parameters compared with MobileNets. However, larger models do not necessarily trigger more errors. In fact, we find that all three DNN models are well-formed: their computation graphs, though large, contain many repeated usages of basic DNN operators. Hence, larger computation graphs do not necessarily impose a higher challenge for DL compilers; see further discussions in Sec. 7.

We attribute the promising correctness of XLA to the "trampoline" code generation paradigm, which has been introduced in detail in Sec. 2.1. Using this paradigm implies a succinct compilation pipeline without involving many heavyweight optimization routines. The tradeoff would be dependency and requirement on the kernel libraries offered by the underlying computation platforms.

Table 5. Model mutation time.

Seed Model	Total (hour)	Average (sec)
ResNet18	2.10	15.12
VGG11	2.79	20.05
MobileNets	2.31	16.63

Processing Time. Our experiments are launched on Intel Xeon CPU E5-2678 with 256GB RAM. Table 5 reports the processing time of mutating models in ONNX format. Note that mutating DNN models in ONNX format generates a considerable number of I/O operations to load models from disk and to dump mutated models back to disk; these operations primarily contribute to the cost. Given all the mutated models, Table 6 further reports the time taken when testing each compiler. We first clarify that when enabling the full optimization (-O3), TVM throws quite a number of compilation failures (see further discussion in Sec. 6.1). As suggested by the developer of TVM, we instead use a lower optimization level (-O2) and re-run the experiments. Overall, our observation shows that TVM and XLA are generally slower than Glow when compiling a DNN model. In particular, TVM -O2 takes over ten days to compile all the test cases, whereas XLA needs over seven days. In contrast, the prediction time of DNN executables are generally rapid: XLA-generated executables only take around 0.1 seconds to make one prediction.

6.1 Characteristics of Compilation Failures

This section elaborates on some errors we have countered that impede compilation. We manually checked all compilation failures of Glow (69 cases) and TVM with full optimization enabled (248 cases). We summarize our findings below.

Table 6. Model compilation time and DNN executable prediction time.

Compiler	Total Compilation Time (hour)	Average Compilation Time (sec)	Total DNN Executable Prediction Time (sec)	Average DNN Executable Prediction Time (ms)
Glow	31.15	86.87	812	650.28
TVM -O2	248.18	631.89	12,492	8,834.00
TVM -O3	8.28	175.00	98	580.00
XLA	182.62	464.00	159	112.46

TVM. When enabling the full optimization of TVM, we encountered a large volume of (simplified) compilation failures as follows:

```
Incompatible broadcast type TensorType([4, 16, 4, 4, 4]) and TensorType([1, 0, 1, 4, 4])
```

The above issue seems to be due to the layout optimization passes (e.g., the AlterOpLayout pass [81]) of TVM, which replace the layout of certain DNN operators in an analysis-friendly form. When broadcasting the re-formed layout to neighboring operators, certain inconsistencies seemingly occurred and terminated the compilation. In fact, all TVM compilation errors yield identical or highly similar error messages. As suggested by the TVM developer, we lower the optimization level from full optimization (-O3) to -O2 to bypass this compilation error.

Glow. When compiling the Sum operator, Glow (with full optimization enabled) seems to optimize the inputs of a Sum operator into an inconsistent format, and terminate the compilation. Considering the following comparison,

```
// Sum in the DNN model of ONNX format
  Inputs0 : float<1 x 1 x 1>
  Inputs1 : float<1 x 2 x 2>
  Inputs2 : float<1 x 2 x 2>
  Result  : float<1 x 2 x 2>
```

```
// Sum in Glow IR after optimization
  Inputs0 : float<1 x 1 x 1>
  Inputs1 : float<1 x 2 x 2>
  Inputs2 : float<1 x 2 x 2>
  Result  : float<3 x 1 x 1>
```

where Glow complains about a mismatch between the first and second inputs' shape, and throws an error message "Mismatching dimension 1 for input 0 and 1". In contrast, in the input model, the first and second operators of Sum have valid and consistent shapes. We suspect this is very likely due to dimension-related optimizations.

Table 7. Buggy code fragments found in Glow (3rd to 8th rows) and TVM (9th to 10th rows). We report the number of error-triggering inputs due to each buggy code fragment, e.g., 21 compilation defects are due to the same root cause — buggy Memory Allocate. We also characterize each buggy code fragment in terms of its role in the compilation pipeline (see Sec. 2.1 for the compilation pipeline).

	Compilation Pipeline				**Total**
	IR Connectivity	**Graph Optimization**	**Low-Level Optimization**	**Others**	
DKKC8 Optimization	0	0	73	0	73
Dead Code Elimination	0	62	0	0	62
ShareBuffers	21	0	0	0	21
Memory Allocate	0	0	8	0	8
Numeric Accuracy	0	0	0	69	69
Total	21	62	81	69	233
Numeric Accuracy	0	0	0	202	202
Total	0	0	0	202	202

6.2 Characteristics of Compilation Defects

This section studies the characteristics of exposed compilation errors. Table 7 classifies all the compilation errors into four categories in accordance with the standard DNN compilation pipeline

introduced in Sec. 2.1. As mentioned in Sec. 4.4, we form a group of analysts to classify the findings manually. This ensures the accuracy of our research. Nevertheless, we admit the difficulties of manual debugging, given the enormous number of faults discovered and the requirement to debug the compilation process in order to understand each problem. As a result, we do classification *at our best effort*. Overall, for Glow, we successfully localize buggy procedures hidden in the IR conversion, high-level (graph-level) optimizations, as well as low-level optimizations. We also find 69 errors that are due to accuracy deviation during the computation of large numeric numbers. As for TVM, with manual investigation, we confirm that all 202 compilation errors are due to large numeric numbers computed by the DNN models.

All of these findings are logic bugs, causing erroneous outputs rather than DL compilers throwing exceptions. The "ShareBuffers" case, leading to 21 incorrect compilations, denotes a bug from the IR conversion phase; we will discuss this case in Sec. 6.2.1. The rest findings are from the high-level or low-level optimization phases. The "Dead Code Elimination" case represents a bug in optimizing the high-level graph IR (see Sec. 6.2.2). In contrast, DKKC8 optimization case denotes a failure when optimizing the low-level IR code, particularly the Conv layer (Sec. 6.2.3). We also find buggy memory allocation in Glow during optimizing low-level IR code; see Sec. 6.2.4. For the other cases which are due to numeric accuracy deviation, we put them in the "Others" category. While it is not surprising that numeric accuracy deviations frequently exist in DNN machine code, we find common DNN operators that can amplify accuracy deviations. This indicates that numeric accuracy deviations might be worse than we expected before. See relevant discussions in Sec. 6.2.5 and comments from the Glow developers regarding this point in Sec. 6.3.

```
1. %Add_2545      = allocactivation  { Ty: float<4 x 1 x 1 x 64>}
2. %Add_2545_res  = elementadd @out %Add_2545, @in %var1, @in %var2
3. %transpose_24  = tensorview @in %Add_2545 { Ty: float<4 x 64 x 1 x 1>}
4.    . . .
5. %Mul_3933_res  = elementmul @out %Mul_3933, @in %Add_2545, @in %Add_2545
6.    . . .
7. %Add_10273_res = elementadd @out %Add_10273, @in %var3, @in %transpose_24
```

(a) Glow IR before ShareBuffers optimization.

```
1. %Add_2545      = allocactivation  { Ty: float<4 x 1 x 1 x 64>}
2. %Add_2545_res  = elementadd @out %Add_2545, @in %var1, @in %var2
3. . . .
4. %Mul_3933_res  = elementmul @out %Add_2545, @in %Add_2545, @in %Add_2545
5. . . .
6. %Add_2545      = tensorview @in %Add_2545 { Ty: float<4 x 64 x 1 x 1>}
7. %Add_10273_res = elementadd @out %Add_10273, @in %var3, @in %Add_2545
```

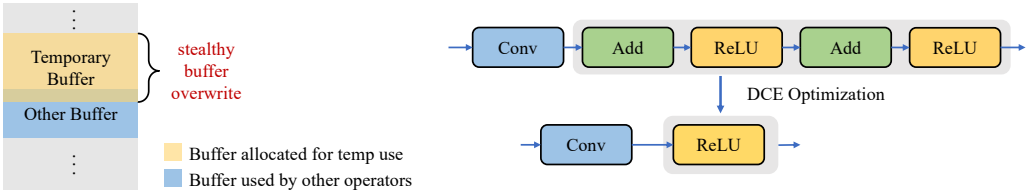
(b) Glow IR after ShareBuffers optimization.

Fig. 8. Bug_1 illustration. Note that `XXX_res` serves as a flag, indicating if the statement is successfully executed.

6.2.1 Bug_1: ShareBuffers. The bug is caused by the ShareBuffers optimization of Glow. In general, each time when allocating memory for storing the current operator's output, this optimization strategy reuses the memory buffer of a previous operator instead of allocating a new memory buffer. Considering Fig. 8, where we dump the IR code before and after the ShareBuffers pass of Glow. Before the ShareBuffers optimization pass, the output buffer of `Add_2545` is used by `Mul_3933` (line 5 in Fig. 8(a)). `Add_2545`, through `transpose_24`, is further used by `Add_10273` (line 7 in Fig. 8(a)). After the optimization, the operator `Mul_3933` reuses the existing buffer by writing its output to the buffer of `Add_2545` (line 4 in Fig. 8(b)). Note that this buffer was previously used to store the

output of DNN node `Add_2545`. Glow leverages this optimization to avoid allocating a new buffer for the output of `Mul_3933` on line 4 in Fig. 8(b).

However, we find that this optimization breaks the topological connectivity of the DNN nodes. According to the optimized IR in Fig. 8(b), the buffer `Add_2545` will be taken as the input of node `Add_10273` (line 7 in Fig. 8(b)). However, given that buffer `Add_2545` has been used by the output of `Mul_3933` on line 4 in Fig. 8(b), `Add_10273` is incorrectly fed with the output of `Mul_3933`. By referring to the computation graph of the reference model in ONNX format, `Add_10273` should accept the output of `Add_2545`, instead of the output of `Mul_3933`. It is seen that in the IR code, incorrect connectivity is introduced by the ShareBuffers optimization.



(a) Bug_2. (b) Bug_3.

Fig. 9. Illustration of bug_2 and bug_3.

6.2.2 Bug_2: Memory Allocate. The intermediate results (buffers) of the model inference are stored in a huge coalesced memory region. Glow determines the actual memory area for each buffer at compile time. The memory allocator used by the low-level code is responsible for allocating all the buffers within a single coalesced region. Our manual study shows that input/output buffers belonging to different operators are reasonably separated and do not overlap. However, note that certain memory regions other than input and output buffers may be used internally and temporarily by DNN operators. Careless writing to the temporary buffer may overwrite the buffer being used by other operators. In this case, a Conv layer, when writing to its temporary storage, is seen to cause a severe memory overwrite issue. As shown in Fig. 9(a), it tampers buffers of other operators and results in erroneous outputs.

6.2.3 Bug_3: Dead Code Elimination. Glow heavily performs graph-level optimizations to simplify the high-level (graph) IR of the target DNN model. Note that at this stage, nodes on the computation graph are high-level DNN operators like Conv, Add, and ReLU. Dead Code Elimination (DCE) is one frequently-applied graph optimization pass at this stage. This optimization removes computations whose results or side-effects do not influence the model predictions. However, we found that Glow's implementation of DCE optimization was buggy, which is seen to have caused some live nodes to be deleted incorrectly from time to time. We illustrate our observation on a buggy case in Fig. 9(b), where three operators are optimized out, bridging the Conv operator directly to the ReLU and inducing subsequent incorrect compilation.

6.2.4 Bug_4: DKKC8 Optimization. Glow applies the layout optimization on Conv. This optimization tries to optimize the standard form of Conv using a customized filter memory layout. Following the convention, the default layout format of Conv is usually referred to as *DKKC*, where D is the number of output channels, C is the number of input channels, and K is the kernel size. An optimization pass named *DKKC8* changes the layout to $[D/8, K, K, C, 8]$ to smoothly leverage the Intel Streaming SIMD Extensions (SSE) instructions to speed up the computation. As clarified in Sec. 4.4, our manual investigation compares the input/output of each Conv operator. At this step,

we successfully localized the output of an optimized Conv operator, which is inconsistent with the corresponding Conv operator in the DNN model running in ONNX runtime (`onnxruntime`).

6.2.5 Numeric Accuracy Deviations. With manual investigation, we find that 271 (69 + 202) erroneous compilations are due to numeric accuracy deviations. These test cases, although exhibiting inconsistent outputs, are not due to logic bugs in DL compilers. Recall as clarified in Sec. 3.2.3, compiler optimizations may introduce small numeric accuracy deviations in DNN executables. We find that the inconsistency originates from small numeric deviations accumulated to noticeably large values during mutation iterations. Therefore, we regard them as false positives.

Overall, given the representation of floating-point numbers in machine instructions and compiler optimizations, the code generated by DL compilers will likely contain precision losses. Our empirical findings and experiences show that such precision loss is usually less than 10^{-6}, which can be ignored. However, when *iteratively* mutating large-scale DNN models to test DL compilers, certain small numeric accuracy driftings are amplified considerably as the complexity of mutated models increases. Recall in Table 3, the largest VGG11 model after mutation can have 468× more nodes and edges compared with the seed VGG11 model (VGG11 already denotes one of the largest DNN models).

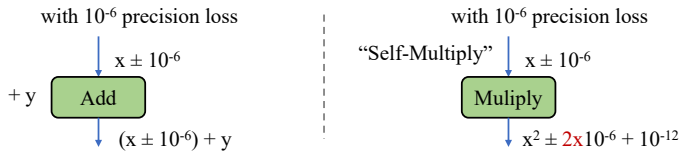


Fig. 10. Clarifying numeric accuracy deviation.

Even worse, some of our inserted DNN operators, e.g., the "self-multiply" node shown on the top left of Fig. 5, can easily amplify the accuracy deviations. Consider Fig. 10, where the output x of a DNN operator is associated with a precision loss of 10^{-6}. While the precision deviation is retained when being processed by a DNN Add operator, the deviation is amplified by $2x$ times when feeding x to a DNN Multiply operator. For our current implementation, to avoid amplifying precision losses, users have to resort to entirely excluding DNN Multiply operators in the construction of mutated models. Nevertheless, doing so may neglect opportunities to stress DL compilers and to expose defects. Overall, users can decide between better stressing DL compilers (by using Multiply operators) and imposing fewer false positives (by excluding Multiply operators).

False Positives. We clarify that such numeric accuracy deviation is a common concern in DNN executables; our study at this step shows that numeric accuracy can be stealthily drifted and even amplified in DNN executables. On the one hand, we consider these findings to be *not* due to logic bugs in DL compilers; rather, they impose a general challenge to DL compilation techniques, whose solution is unclear at this stage. We thus deem our findings at this step as "false positives." On the other hand, it might not be inaccurate to assume that such numeric accuracy deviations denote undefined and unwanted behaviors, which stealthily undermine the computation of DNN executables. Instead of letting this issue "silently" occur, we see it as a demanding need to design techniques that can monitor the execution of DNN machine code and alter users about noticeable deviations in terms of numeric accuracy. Sanitization techniques might be potentially promising to deliver online monitoring [70]. In short, we leave it as one future work to explore designing proper sanitizer checks to alter large numeric accuracy deviations.

6.3 Confirmation with Developers

To seek prompt confirmation and insights into our results, we have reported our findings to the developers, including bug-triggering DNN models, localized DNN operators, and relevant compilation/optimization modules in the DL compilers. The Glow developer was responsive in clarifying our findings regarding numerical number differences. He shares the same opinion with us that numerical number deviations are generally hard to eliminate, which should be treated as "false positives" (as we did in Sec. 6.2.5). To quote him:

> *"I wouldn't be surprised if we had some potential numeric differences when removing a layer of the model. Especially when considering moving between different datatypes/precisions, removing some layer may end up causing additional conversions that can change numerics of a graph, due to a change in how graph optimizations impact the final compiled model."*

We also quote the feedback from TVM author's feedback regarding the exposed compilation failures below:

> *"Seems like an opt_level=3 pass messes up with the shapes. It might be AlterOpLayout at first glance, [...]"*

At the time of writing, we are waiting for the response from TVM and Glow on the logic errors found by MT-DLComp. Overall, given the high risk of our findings, we expect responsible and prompt confirmation from the developers and to incorporate our findings into their scheduled development pipeline.

7 DISCUSSION

Our study demonstrates that real-world DL compilers exhibit a high level of engineering quality. Nevertheless, our systematic testing successfully reveals a large number of logic flaws and compilation failures. We further identified several root causes in DL compilers and presented discussions accordingly. This section compares the effectiveness of our two mutation schemes. We also discuss some limitations and prospective enhancements of our testing pipeline.

Table 8. Result overview of comparision between UOC and PIOC. We generate $3 \times 5 \times 1000$ mutated models by solely using UOC or PIOC. We then count the number of error-triggering models found by these two mutation methods respectively.

DL Compiler	#Errors Exposed by UOC		#Errors Exposed by PIOC	
	Compilation Failures	Logic Error	Compilation Failures	Logic Error
TVM -O2	0	0	0	0
Glow	9	136	363	183
XLA	0	0	0	0
Total	9	136	363	183

Comparing UOC and PIOC. Our evaluation launched in Sec. 6 iteratively mutates a seed model, where for each iteration, we randomly select a method from either the UOC or PIOC schemes. At this step, we launch further assessment to benchmark and compare the effectiveness of these two schemes. To that end, we mutate seed DNN models with the same setup as in Sec. 6, except that we solely employ UOC or PIOC in each series of iterative mutation instead of randomly selecting one scheme. In accordance with our previous experiments (see Sec. 5), we use in total $1,500 \times 2$ mutated models (half are mutated using only UOC and the rest are mutated using only PIOC) to test DL compilers and count the exposed errors.

The findings are shown in Table 8. Out of 1,500 test cases in total, UOC triggered 9 compilation failures, whereas PIOC triggered 363. For the rest of the test cases which can be compiled successfully,

136 mutated models generated via UOC exhibit inconsistent output with their corresponding seed DNN models, while there are 183 in PIOC. Both methods didn't find any bug in XLA and TVM. Among all the discovered deviant-output cases, we manually confirmed that there is only one false positive in UOC and PIOC respectively, whose root cause is floating-point drifting. This means that nearly all the erroneous test cases with inconsistent output are due to compiler logic bugs. The experiment results show that PIOC can trigger more errors than UOC, indicating that PIOC may be more effective to stress compilers. Nevertheless, PIOC comes at a higher cost for mutation than UOC, since profiling stages are required for this scheme, which are time-consuming. We report that mutating for 1,000 iterations using the PIOC scheme takes approximately 15 hours in our experiment. In comparison, UOC takes less than one minute. Additionally, we find that the total number of errors exposed by PIOC or UOC alone is smaller than that generated by combining them, showing that combining PIOC and UOC can have a synergistic impact, increasing the complexity of DNN models and stressing DL compilers. However, the number of false positives, due to amplified precision losses, is lower when PIOC and UOC are used alone than when they are combined. When deciding whether to combine PIOC and UOC, users might decide between increased bug exposure and increased false positives.

Quality of Test Seeds. We employ three DNN models, VGG11, ResNet18, and MobileNets, as the testing seeds. These models are frequently used in daily DL tasks. Two of these seeds, VGG11 and ResNet18, are large-scale CV models, whereas the third is frequently used on low-cost devices. Our intuition is that by stressing DL compilers with real-world models, particularly large models like VGG11 and ResNet18, we can effectively reveal compiler faults. However, our observational and manual tests during evaluation illustrates enhancement opportunities regarding test seed; employing these real-world DNN models, though fruitful in detecting bugs, may not be the best practice. The observation is that CV models such as VGG11 and ResNet18, despite their large number of DNN operators and millions of parameters, are formed on the basis of some elementary DNN operators (e.g., the Conv operator). We view the computation graphs of these models are "well-formed," in the sense that their computation graphs contain *repeated* patterns of connected elementary DNN operators. Note that this may jeopardize the efficiency of testing, since we may squander mutation efforts stressing likely identical compilation and optimization procedures in the DL compilers.

Overall, using repetitive DNN operators to form computation graphs are commonly seen in daily large-size DNN models. We also clarify that utilizing "uncommon" DNN models are not expected to help solve the problem either, since most uncommon DNN models are also built by repeatedly stacking elementary DNN operators. In summary, we envision the possibility of synthesizing our own seed models, which should particularly encompass a variety of distinct DNN operators on a model's computation graph. We leave it as a future task to explore improving seed diversity and enhancing testing effectiveness.

Guided Mutation on the DNN's Computation Graph. As mentioned in Sec. 4.2.4, when iteratively mutating a seed model, MT-DLComp randomly selects a place on the computation graph to insert some additional DNN operators each time. Along with random mutation, we anticipate several heuristic-based ways for increasing the diversity of mutated models, which could likely improve the effectiveness of testing DL compilers. For example, we envision that the Markov Chain Monte Carlo (MCMC) algorithm and adaptive random testing (ADT) [10] can be used to direct MT-DLComp toward insertion locations that maximize coding diversity [39]. To do so, for each new variant $M' = T(M)$ (where T is our MR), we compute a distance metric between M' and its seed M. M' with a greater distance will be kept for further mutation, whereas M' showing a small distance could be discarded. In comparison to random mutation, such heuristic-based approaches can likely

generate more diverse mutations and detect more faults [10, 39]. We leave it as one future work to incorporate such heuristic-based methods to supplement MT-DLComp.

Limitations of Employing Metamorphic Testing. This research employs metamorphic testing to test DL compilers. From a holistic view, our technique is based on the idea of mutating seed DNN models to generate semantically-equivalent variants and asserting output consistency among the compiled DNN executables. Nevertheless, even if all variants generate outputs that are *consistent* with the seed model, we cannot ensure that the outputs are *correct*. Consider the case where a DL compiler contains a defect and incorrectly compiles the seed model. If that DL compiler commits the exact same error when compiling variants mutated from the seed, all variants will presumably produce the same but incorrect output as the seed model. Due to the fact that our approach tests only the output *consistency*, we cannot discover this circumstance in which all mutated models behave consistently albeit incorrectly. On the other hand, we clarify that this limitation of metamorphic testing does not affect our findings: we manually confirmed that all seed models were appropriately built and made correct predictions (w.r.t. our prepared model inputs) by the DL compilers.

Future Work: Uncovering Performance Bugs in DL Compilers and Kernel Libraries. As introduced in Sec. 4, our study focuses on finding logic bugs in DL compilers that generate *malfunctional executables*. A DL compiler is also anticipated to generate efficient code [40] on various hardware backends. Nevertheless, it is seen that the compiled executables may perform poorly under certain circumstances, such as when a Conv kernel has an edge-case batch size [11]. Overall, developing techniques for detecting bugs in DL compilers that generate *ill-performed executables* is a worthwhile area of future research. Plus, DL compilers like XLA and NNFusion extensively employ kernel libraries manually written by experts to deliver speedy computations of elementary DNN operators. It is worthwhile studying if unexpected performance degradation exists in these low-level kernel libraries.

8 RELATED WORK

We have reviewed compilation techniques of DNN models in Sec. 2.1. In this section, we review recent efforts in assessing DNN models and infrastructures via software testing.

Testing DNN Models. Previous studies primarily focus on testing DNN-based image classification models, which holistically describe an image [16, 51, 55, 59, 79, 86, 91, 92, 97, 103]. Some recent studies also explore testing object recognition models [66, 78, 87]. In addition, natural language processing (NLP) models and downstream applications like machine translation [6, 24, 26, 27, 74], sentiment analysis [19, 44, 82], visual question answering (VQA) [93], and natural language interface of databases (NLIDB) [43] have also been tested. From the perspective of network architectures, while most testing target CNN models (as they primarily drive image analysis), recurrent neural networks (RNN) and reinforcement learning models are also tested [13, 20, 32, 57, 83].

Testing DL Infrastructures. In addition to testing DNN models, existing works have been proposed to test machine learning libraries [14–16, 67, 72]. With the flourishing development of DL infrastructures, differential testing has been used [23, 60] toward multiple DL libraries to capture their inconsistencies. Wang et al. [88] propose techniques to mutate DL models to better invoke corner execution paths of DL libraries. Empirical studies have also been conducted to analyze bug characteristics of DL software and libraries [22, 34, 99]. Floating-point numbers and their potentially-induced precision-related errors have also been studied in DL libraries using software testing and empirical efforts [98, 100]. Shen et al. [68] launched empirical studies to characterize bugs of DL compilers; they primarily analyze bug reports collected from places like GitHub bug fix pull requests and developer forums. Our work, for the first time, proposes an automated and

comprehensive framework to test DL compilers and expose compiler bugs that can stealthily generate incorrect DNN executables.

Testing Oracles. Preparing a validation dataset with labelled data is the traditional way to benchmark DNN models. However, standard datasets, with limited labelled data, can only reflect the hold-out accuracy [59, 63, 74, 96]. Many corner cases (e.g., a snowy traffic scene) can be rarely included. The majority of existing testing efforts primarily leverage testing schemes carrying implicit oracles. For example, metamorphic testing can determine if a DNN model responds consistently to mutated inputs [16, 79, 87, 97]. Similarly, differential testing [59, 79] evaluates if a group of DNN models make consistent predictions under the same inputs. Domain-specific oracles for model robustness [80], fairness [17, 104], and interpretability [28, 78] have also been proposed.

9 CONCLUSION

We conducted a systematic investigation on the compilation correctness of prominent DL compilers. We offer MT-DLComp, a testing framework based on metamorphic testing to perform functionality-preserving mutations on DNN models. Our comprehensive study of four prominent industry-strength DL compilers identified a number of compilation errors. Additionally, we elaborated on our findings and outlined the key takeaways from this study. This work may serve as a roadmap for researchers and users interested in utilizing and improving DL compilers.

ACKNOWLEDGEMENT

We thank anonymous reviewers and our shepherd, Edith Cohen, for their valuable feedback. We also thank Xiaofei Xie and Lei Ma who provide feedback on this project. The research was supported in part by a RGC ECS grant under the contract 26206520.

REFERENCES

[1] 2021. ONNX. https://onnx.ai/.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[3] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[4] Amazon. 2021. Amazon SageMaker Neo uses Apache TVM for performance improvement on hardware target. https://aws.amazon.com/sagemaker/neo/.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[6] Jialun Cao, Meiziniu Li, Yeting Li, Ming Wen, and Shing-Chi Cheung. 2020. SemMT: A Semantic-based Testing Approach for Machine Translation Systems. *arXiv preprint arXiv:2012.01815* (2020).

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[8] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018), 3389–3400.

[9] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong ....

[10] Tsong Yueh Chen, Hing Leung, and IK Mak. 2004. Adaptive random testing. In *Annual Asian Computing Science Conference*. Springer, 320–329.

[11] choi. 2020. TVM Performance Degradation. https://discuss.tvm.apache.org/t/performance-has-been-too-slow-since-the-tvm-update/5865/7.

[12] Al Danial. [n. d.]. CLOC. https://goo.gl/3KFACB.

[13] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Jianjun Zhao, and Yang Liu. 2018. Deepcruiser: Automated guided testing for stateful deep learning systems. *arXiv preprint arXiv:1812.05339* (2018).

[14] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing Probabilistic Programming Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*.

[15] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 729–739.

[16] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying Implementation Bugs in Machine Learning Based Image Classifiers Using Metamorphic Testing. In *ISSTA*.

[17] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*. 214–226.

[18] Michael J Flynn. 1972. Some computer organizations and their effectiveness. *IEEE transactions on computers* 100, 9 (1972), 948–960.

[19] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *ACM ESEC/FSE*. ACM, 498–510.

[20] Jianmin Guo, Yue Zhao, Xueying Han, Yu Jiang, and Jiaguang Sun. 2019. Rnn-test: Adversarial testing framework for recurrent neural network systems. *arXiv preprint arXiv:1911.06155* (2019).

[21] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE transactions on computer-aided design of integrated circuits and systems* 37, 1 (2017), 35–47.

[22] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 810–822.

[23] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 486–498.

[24] Shashij Gupta, Pinjia He, Clara Meister, and Zhendong Su. 2020. Machine translation testing via pathological invariance. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 863–875.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[26] Pinjia He, Clara Meister, and Zhendong Su. 2020. Structure-invariant testing for machine translation. In *ICSE*.

[27] Pinjia He, Clara Meister, and Zhendong Su. 2021. Testing Machine Translation via Referential Transparency. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 410–422.

[28] Bernease Herman. 2017. The promise and peril of human evaluation for model interpretability. *arXiv preprint arXiv:1711.07414* (2017), 8.

[29] SA Hex-Rays. 2014. IDA Pro: a cross-platform multi-processor disassembler and debugger.

[30] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[31] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[32] Wei Huang, Youcheng Sun, Xiaowei Huang, and James Sharp. 2019. testrnn: Coverage-guided testing on recurrent neural networks. *arXiv preprint arXiv:1906.08557* (2019).

[33] Texas Instruments. 2021. The AM335x microprocessors support TVM. https://software-dl.ti.com/processor-sdk-linux/esd/docs/latest/linux/Foundational_Components/Machine_Learning/tvm.html.

[34] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.

[35] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[37] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–.

[38] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*.

[39] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*.

[40] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[41] Zhibo Liu and Shuai Wang. 2020. How far we have come: Testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 475–487.

[42] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 881–897.

[43] Pingchuan Ma and Shuai Wang. 2022. MT-Teql: Evaluating and Augmenting Neural NLIDB on Real-world Linguistic and Schema Variations. In *PVLDB*.

[44] Pingchuan Ma, Shuai Wang, and Jin Liu. 2020. Metamorphic Testing and Certified Mitigation of Fairness Violations in NLP Models. In *IJCAI*. 458–465.

[45] Microsoft. 2020. onnxruntime. https://github.com/microsoft/onnxruntime.

[46] Microsoft. 2021. Microsoft Linear Algebra Subprograms. https://github.com/microsoft/onnxruntime/tree/master/onnxruntime/core/mlas.

[47] Timothy Prickett Morgan. 2020. INSIDE FACEBOOK'S FUTURE RACK AND MICROSERVER IRON. https://www.nextplatform.com/2020/05/14/inside-facebooks-future-rack-and-microserver-iron/.

[48] MT-DLComp. 2021. MT-DLComp. https://github.com/Wilbur-Django/Testing-DNN-Compilers.

[49] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.

[50] Varsha Nair, Moitrayee Chatterjee, Neda Tavakoli, Akbar Siami Namin, and Craig Snoeyink. 2020. Fast Fourier Transformation for Optimizing Convolutional Neural Networks in Object Recognition. arXiv:2010.04257 [cs.CV]

[51] Shin Nakajima and Tsong Yueh Chen. 2019. Generating biased dataset for metamorphic testing of machine learning programs. In *IFIP-ICTSS*.

[52] Nvidia. 2021. NVVM IR. https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html.

[53] NXP. 2020. NXP uses Glow to optimize models for low-power NXP MCUs. https://www.nxp.com/company/blog/glow-compiler-optimizes-neural-networks-for-low-power-nxp-mcus:BL-OPTIMIZES-NEURAL-NETWORKS.

[54] OctoML. 2021. OctoML leverages TVM to optimize and deploy models. https://octoml.ai/features/maximize-performance/.

[55] Augustus Odena and Ian Goodfellow. 2018. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. *arXiv preprint arXiv:1807.10875* (2018).

[56] ONNX. 2021. ONNX Zoo: A collection of pre-trained, state-of-the-art models in the ONNX format. https://github.com/onnx/models.

[57] Qi Pang, Yuanyuan Yuan, and Shuai Wang. 2021. MDPFuzzer: Finding Crash-Triggering State Sequences in Models Solving the Markov Decision Process. *arXiv preprint arXiv:2112.02807* (2021).

[58] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[59] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785

[60] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*.

[61] Qualcomm. 2020. Qualcomm contributes Hexagon DSP improvements to the Apache TVM community. https://developer.qualcomm.com/blog/tvm-open-source-compiler-now-includes-initial-support-qualcomm-hexagon-dsp.

[62] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[63] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4902–4912. https://doi.org/10.18653/v1/2020.acl-main.442

[64] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[65] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.

[66] Jinyang Shao. 2021. Testing Object Detection for Autonomous Driving Systems via 3D Reconstruction. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 117–119.

[67] Arnab Sharma and Heike Wehrheim. 2019. Testing machine learning algorithms for balanced data usage. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 125–135.

[68] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 968–980.

[69] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[70] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1275–1295.

[71] Ezekiel Soremekun, Sakshi Udeshi, and Sudipta Chattopadhyay. 2020. Astraea: Grammar-based Fairness Testing. *arXiv preprint arXiv:2010.02542* (2020).

[72] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-implementation testing of supervised learning software. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.

[73] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *OOPSLA*.

[74] Zeyu Sun, Jie M Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. 2020. Automatic testing and improvement of machine translation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 974–985.

[75] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[76] TensorFlow. 2019. XLA: Optimizing Compiler for TensorFlow. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[77] Tensorflow. 2020. Tensorflow backend for ONNX (Open Neural Network Exchange). https://pypi.org/project/onnx-tf/.

[78] Yongqiang Tian, Shiqing Ma, Ming Wen, Yepang Liu, Shing-Chi Cheung, and Xiangyu Zhang. 2021. To what extent do DNN-based image classification models make unreliable inferences? *Empirical Software Engineering* 26, 5 (2021), 1–40.

[79] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars *(ICSE '18)*.

[80] Vincent Tjeng, Kai Xiao, and Russ Tedrake. 2017. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356* (2017).

[81] TVM. 2020. AlterOpLayout. https://tvm.apache.org/docs/api/python/relay/transform.html.

[82] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated Directed Fairness Testing *(ASE)*.

[83] Jonathan Uesato, Ananya Kumar, Csaba Szepesvari, Tom Erez, Avraham Ruderman, Keith Anderson, Nicolas Heess, Pushmeet Kohli, et al. 2018. Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures. *arXiv preprint arXiv:1812.01647* (2018).

[84] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[85] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[86] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. 2019. Adversarial Sample Detection for Deep Neural Network Through Model Mutation Testing *(ICSE)*.

[87] Shuai Wang and Zhendong Su. 2020. Metamorphic Object Insertion for Testing Object Detection Systems. In *ASE*.

[88] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799.

[89] Sally Ward-Foxton. 2021. Google and Nvidia Tie in MLPerf; Graphcore and Habana Debut. https://www.eetimes.com/google-and-nvidia-tie-in-mlperf-graphcore-and-habana-debut.
[90] Xilinx. 2020. Xilinx support TVM on DPU. https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/deploying_running.html.
[91] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2021. Enhancing Deep Neural Networks Testing by Traversing Data Manifold. *arXiv preprint arXiv:2112.01956* (2021).
[92] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2021. You Can't See the Forest for Its Trees: Assessing Deep Neural Network Testing via NeuraL Coverage. *arXiv preprint arXiv:2112.01955* (2021).
[93] Yuanyuan Yuan, Shuai Wang, Mingyue Jiang, and Tsong Yueh Chen. 2021. Perception Matters: Detecting Perception Failures of VQA Models Using Metamorphic Testing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 16908–16917.
[94] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2012. Alphaz: A system for design space exploration in the polyhedral model. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 17–31.
[95] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.
[96] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).
[97] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *ASE*.
[98] Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. 2021. Predoo: precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 400–412.
[99] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs *(ISSTA 2018)*.
[100] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 826–837.
[101] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 863–879.
[102] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.
[103] Husheng Zhou, Wei Li, Yuankun Zhu, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. Deepbillboard: Systematic physical-world testing of autonomous driving systems *(ICSE)*.
[104] Indre Zliobaite. 2017. Fairness-aware machine learning: a perspective. *arXiv preprint arXiv:1708.00754* (2017).

Received October 2021; revised December 2021; accepted January 2022